Inherent Risks in Object-Oriented Development©

Dr. Peter Hantos
The Aerospace Corporation

Object orientation has been in existence since the late 1970s. During the 1990s, however, on the basis of various claims that it was a dramatic, new software engineering approach, object-oriented software development became pervasive. Currently, most new software projects use object-oriented (OO) techniques to various extents. The persistence of schedule slips and cost overruns, particularly in the case of the development of large-scale, software-intensive systems, raises the need for revisiting the basics and exploring the inherent risks that OO technology might contribute to the overall risk profile of a project. In this article, Bertrand Meyer's classic OO technology concepts are mapped into Barry Boehm's Top 10 methodology-neutral software risks to illustrate potential areas of exposure. Recent developments in OO technology, such as Java, Use Cases, or the Unified Modeling Language fit well into this framework and are included as examples. The systematic approach introduced will allow project managers to better understand the cost/benefit aspects of applying OO technology, and to align their project management strategies more successfully with the organization's business goals.

In this article, the term *object-oriented* (OO) technology refers to OO development processes and methods, object-related standards, and associated products and tools from third-party vendors. Enterprises that develop software are looking to OO as a means to achieve their strategic business objectives. They expect that OO will enable them to build complex systems of superior quality with reduced development time and costs, while providing long-term benefits such as maintainability, reusability, and extensibility.

If, in fact, OO has been in use for a relatively long period, then why is it still necessary to explore OO-specific risks? The simple answer can be found in R.L. Glass' 2002 article [1]. According to Glass, the introduction of a technology is no guarantee of effective use. Similar to OO, other technologies such as fourth-generation languages and computer-assisted software engineering tools were introduced with great fanfare, but once the technology was more thoroughly understood, the benefits turned out to be far more modest than originally claimed.

Also, OO risks are not the same as those associated with the introduction of any new technology. With respect to paradigm scope, complexity, and depth, OO has far-reaching consequences. For the project manager, the decision is not simply whether to apply OO to a particular project: The use of OO permeates all aspects of development. Based on business priorities, project managers must determine the desired penetration of OO concepts, the optimal insertion order, and whether the replacement of legacy languages and tools is justified.

Object-Oriented Technology

In his 1995 book [2], Bertrand Meyer provides a sound overview of OO fundamentals. According to Meyer, software construction embracing OO is structured around the following concepts[1].

- **M1:** A unique way to define architecture and data structure instances.
- **M2:** Information hiding through abstraction and encapsulation.
- **M3:** Inheritance to organize related elements.
- **M4:** Polymorphism to perform operations that can automatically adapt to the type of structure they operate on.
- **M5:** Specialized analysis and design methods.
- **M6:** OO languages.
- **M7:** Environments that facilitate the creation of OO systems.
- **M8:** *Design by Contract*, a powerful technique to circumvent module boundary and interface problems.
- **M9:** Memory management that can automatically reclaim unused memory.
- **M10:** Distributed objects to facilitate the creation of powerful distributed systems.
- **M11:** Object databases to move beyond the data-type limitations of relational database management systems.

Please note that this article is not intended to be a tutorial on OO; rather, it will examine risk implications associated with all of these concepts. It is assumed that the reader is familiar with the basics.

Risk Management

Risk management is acknowledged as a critical process of project management, and has received more and more attention since the 1980s. For example, in the Software Engineering Institute-developed[2] process improvement framework, during the transition from the Capability Maturity Model for Software® (SW-CMM®) to CMM Integration℠ (CMMI®), risk management was elevated from a recommended practice to a formal, independent process area. Nevertheless, to accommodate a broader audience, the definitions used in the following discussion are based on IEEE-STD-1540-2001 [3] and not CMMI materials.

Risk is defined as a potential problem, an event, hazard, threat, or situation with undesirable consequences. The non-deterministic nature of risk makes risk management a special challenge for the project manager. During project planning, we might be tempted to try to avoid risks altogether, but relying strictly on avoidance as a risk mitigation technique is usually not adequate. The success of a project depends primarily on the project manager's ability to manage the delicate balance between opportunity and risk. Unfortunately, when all risk goes away, so does opportunity. That is why successful project management practices include risk management, a continuous process for systematically addressing risk throughout the life cycle of a product or service.

According to IEEE-STD-1540-2001, the risk management process consists of the following activities:

1. Plan and implement risk management.
2. Manage the project risk profile[3].
3. Perform risk analysis.
4. Perform risk monitoring.
5. Perform risk treatment.
6. Evaluate risk management processes.

The focus of this article is risk identification, a critical aspect of risk analysis. Risk identification, similar to all other elements of continuous risk management, is

® Capability Maturity Model, CMM, and CMMI are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.
℠ CMM Integration is a service mark of Carnegie Mellon University.

not a one-time activity. Changes in the risk management context and changing management assumptions represent major risk sources, and need to be continuously monitored as well. IEEE-STD-1540-2001 does not prescribe how risks should be identified, but it suggests numerous methods, including the use of risk questionnaires or brainstorming.

A specialized example of a risk questionnaire, to be used in a Java 2 Enterprise Edition (J2EE) environment, is presented in [4]. Most risk questionnaires are the result of some sort of brainstorming effort; in most cases, the authors interviewed experienced project managers about their past projects and, after some filtering and processing, they turn the structured risk statements into questions or checklists. For an example of a systematic approach to develop a checklist, see Tony Moynihan's article [5].

Barry Boehm first published his Top 10 Software Risks in 1989 [6], and presented an updated list in his 1995 software engineering course with surprisingly few modifications that were based on feedback from the University of Southern California's Center for Software Engineering Industrial Affiliate companies. (For a published version of the second list please see [7].) Essentially all items, although sometimes named slightly differently, still represented major risk sources, and the name changes can be attributed to changes in popular terminology and not fundamental root causes.

Identifying OO Risks

Consolidating Boehm's Risk Sources

For the discussion in this article, Boehm's list of Top Ten Software Risks will be consolidated into eight risks as shown in Figure 1. First, items on the 1989 list were crosschecked with the 1995 list. Item No. 5, *gold-plating*, from the 1989 list is clearly a requirements mismatch issue[4]. Finally, on the 1995 list, for the sake of brevity, requirements mismatch has also been combined with user interface mismatch, and commercial off-the-shelf (COTS) issues with legacy software issues since they have many similarities with respect to root causes.

Mapping and Interpreting Meyer's OO Concepts

The objective of the following analysis is to determine what OO concepts and practices are germane to risks viewed as significant by the software community. The key to meeting this challenge is the use of well-proven frameworks to inventory the essential attributes of OO technology and project risks. Boehm's risk identification checklist was chosen because it is well accepted in the software engineering community.

During the mapping process, we examined Boehm's consolidated risk list item by item and identified the corresponding, relevant OO concepts. The results of this mapping are summarized in Figure 2, and a detailed discussion follows in the rest of this article. The dots on Figure 2 represent a relationship between the particular risk item and the corresponding OO concept. Arrows pointing to the risks signify the influence of the selected OO concepts, while arrows pointing to the OO concepts relate to situations where the OO concepts have a risk-mitigating – rather than risk-triggering – effect.

Personnel Shortfalls (Risk B1)

Software development is a highly labor-intensive process, and its success depends primarily on the people in the organization. Beyond well-known organizational and political issues, several OO-specific concerns need to be explored. The most significant concerns are specialized skills and experience, and that is why all OO concepts are connected to this risk item as shown in Figure 2.

The first issue is the right balance between application domain knowledge and OO knowledge. It is difficult to find people skilled in both; hence, the collaboration between project personnel with different skill bases is critical. The second issue is the number and distribution of available people. OO knowledge is relevant for most members of the organization, although not to the same extent. In positions such as managers, architects, developers, and testers, it is important that all personnel have or acquire via training the appropriate OO skills.

For example, to avoid personnel shortfalls, the executives themselves who create, manage, or sponsor the development organization have to understand the essential elements of OO even before staffing starts for a project. While having prior OO experience is an asset for managers, the minimum requirement should be to have a certain level of OO literacy. In fact, Meyer's book, which is used in this analysis, is an excellent tool for this purpose, i.e., educating managers in OO fundamentals[5]. The seeding of all teams with OO mentors is also a good approach to distribute OO domain knowledge and to both jump-start and facilitate OO development.

Not surprisingly, most other sources that have analyzed OO migration have focused on the human dimension as well. Two of the three key items discussed in [4] deal with learning curve and training, and [4] contains further references to other authors addressing the same concern [8, 9].

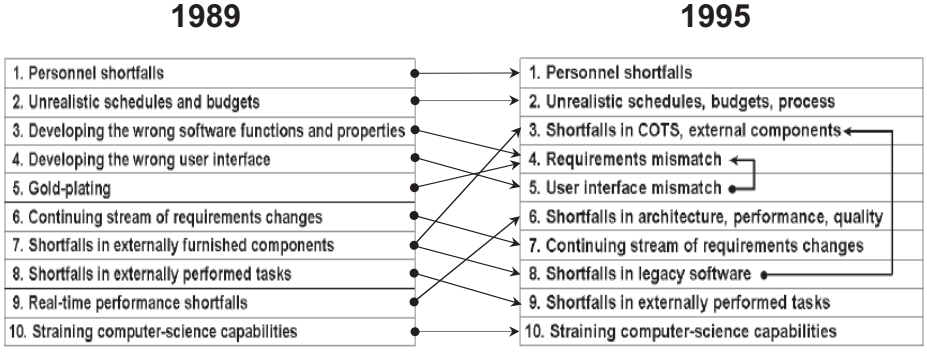
Personnel issues play an important role in the team context as well. OO requires a new way of thinking and moving away from outdated approaches like using functional decomposition for architecting systems or implementing obsolete programming constructs. For teams with a long heritage of using legacy approaches, the paradigm shift is particularly difficult. In fact, sometimes we have observed a quiet, *passive resistance* to OO methods where the people attempted to fake the usage of new methods but at the same time were continuing *business as usual*. A good example for this anomaly is writing *C*-like programs with the use of a *C++* compiler.

Unrealistic Schedules, Budgets, and Process (Risk B2)

Unrealistic expectations, lack of management appreciation for the necessary skills, and the difficulty of the paradigm shift will lead to unrealistic schedules. Similarly, underestimating the time and cost of necessary training would result in unrealistic schedules and budget. Nevertheless, some key OO items specifically contribute to this problem. Based on E. Flanagan's summary [10], most of the time OO projects are introduced on the following grounds:

- OO is better at organizing inherent complexity, and abstract data types make it easier to model the application.

Figure 1: *Consolidating Boehm's Top 10 Software Risks List*



(These statements are building on Concept M1, labeled Architecture and Instances.)

- OO systems are more resilient to change due to encapsulation and data hiding (per concept M2).
- OO design often results in smaller systems because of reuse, resulting in overall effort savings. This higher level of reuse in OO systems is attributed to the inheritance feature (per concept M3).
- It is easier to evolve OO systems over time because of polymorphism (per concept M4).

However, we can also learn from [5] that, particularly when OO is introduced for the first time, expectations might be exaggerated, and frequently the impact of potential costs and risks are minimized to claim maximized payback. For example, it might not be made clear to the sponsoring executives that under certain circumstances it would take several years for just the previously mentioned four benefits of OO to be fully realized. The background of this problem is two-fold. First, building class-libraries is time consuming, or, in case of purchase, they represent a major, up-front investment. Second, to achieve high return on investment, reuse must take place in a very large project or in multiple projects.

One of the side effects of the OO approach is that the design process becomes more important than it was in non-OO projects. Due to encapsulation, data hiding, and reuse, the design complexity moves out of the code space into the design space. The increased design complexity has testing consequences as well. Even if incremental integration is applied, more sophisticated integration test suites need to be created to test systems with a potentially large number of highly coupled objects.

It is also an unfortunate fact that while the OO concepts identified make system comprehension easier during analysis and design, they cause testing and debugging to become more difficult, since now all debugging methodologies and tools have to work with those abstract data types and instances. Those organizations that assume that testing OO is like testing any other software are in for a big surprise. R. Binder makes a powerful case for this argument in his article [11]. According to Binder, it is a common myth that only Black Box[6] testing is needed and OO implementation specifics are unimportant. In reality, OO code structure matters, because inheritance, encapsulation, and polymorphism present opportunities for errors that do not exist in conventional languages. Also, OO has led to new points of view and representations, and the test design techniques that extract test cases from these representations must also reflect the paradigm change.

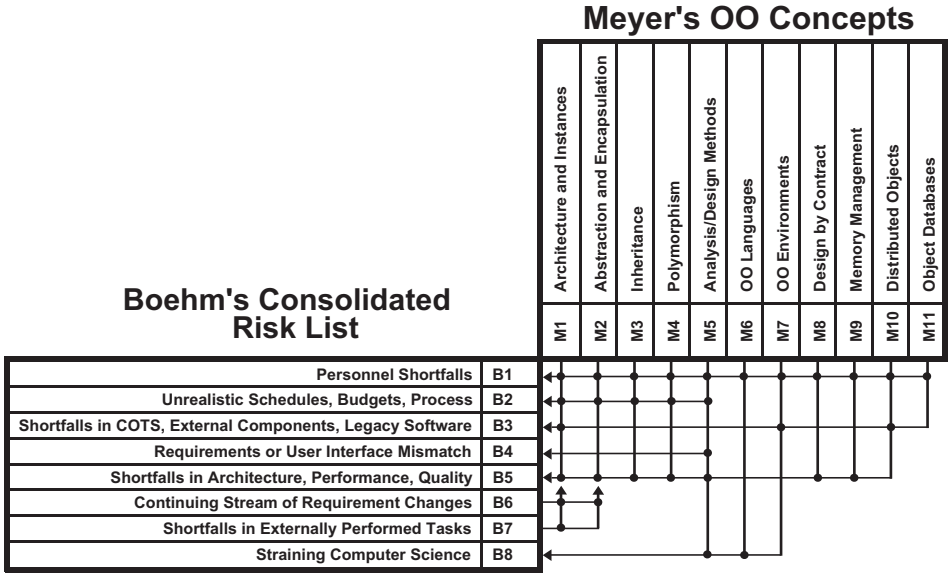


Figure 2: *Mapping Meyer's OO Concepts Into Boehm's Consolidated Risk List*

Shortfalls in COTS, External Components, and Legacy Software (Risk B3)

Using COTS and other externally developed or legacy components in OO presents particular difficulties for structural comprehension and architectural design. These external components, their architecture, interfaces, and documentation are not necessarily consistent with the class and object architecture, communication mechanisms, and view models of the system being developed.

A particular OO problem in this area is the interface of Object Database implementations with traditional Relational Database management systems. The problem may deepen in situations where multiple new technologies merge, for example, in the use of Java-specific object-oriented COTS products (Enterprise Java Beans, Java Message Service, etc.) to develop application services on standard IBM, Sun, and Oracle platforms.

Requirements or User Interface Mismatch (Risk B4)

The OO source of risk is the fact that use cases are used almost exclusively to develop requirements in OO systems. However, use cases only capture functional requirements so additional process steps need to be included to develop and implement quality-related[7], non-functional requirements. An interesting source of Graphical User Interface mismatch is that the Use Case methodology, though well suited for capturing the dynamism of changing screens, is inappropriate for representing screen details.

Shortfalls in Architecture, Performance, and Quality (Risk B5)

This is the area where OO approaches present a controversial impact. Data abstraction, encapsulation, polymorphism, and the use of distributed objects, while increasing architectural clarity, all come with a price: substantial overhead due to the introduced layers of indirection. Unless the system is carefully architected and sound performance engineering practices [12] are implemented from the beginning, satisfying both performance and quality objectives becomes difficult. All of these issues boil down to the earlier mentioned design challenge. Particularly in the case of real-time applications, the system architect must carefully determine the optimal system cohesion. Most real-time performance issues can be resolved if you are willing to suffer increased coupling and the consequent loss of flexibility.

Another sensitive part of OO systems is memory management in general and the implementation of garbage collection in particular. Garbage collection is an integral part of most OO run-time environments. It is a popular technique to ensure that memory blocks that were dynamically allocated by the programmer are released and returned to the free memory pool when they are no longer needed. A typical OO application of this feature is the dynamic creation and destruction of objects. The problem is that in conventional systems, the execution of the main process needs to be interrupted while the garbage collector does its job. This ran-

domly invoked process with variable durations disrupts the real-time behavior of the system.

There are two different approaches to the mitigation of this risk. In the case of real-time OO systems, prudent programming practice should include explicit object creation and destruction to eliminate the dependency on garbage collection. Another solution is the implementation of the garbage collector via multithreading. However, multithreading is a difficult, advanced concept that itself can be the source of numerous risks. For a complete discussion of multithreading implementation pitfalls in Java, see [13].

Finally, a common, OO-related shortfall of architecture pertains to reuse. Most software development organizations moved to OO because engineering managers believed that it would lead to significant reuse. Unfortunately, as the authors of [14] point out, without an explicit reuse agenda and a systematic, reuse-directed software process, most of these OO efforts did not lead to successful, large-scale reuse. Ironically, in some other situations, even the presence of a reuse-driven agenda (platform-based product line development) is no guarantee of success if *reuse* becomes a slogan and senior management expectations are mishandled. In a product line, the participating products share (reuse) architecture and common components, and the implementation of an effective, strategic reuse process becomes a key enabler in achieving low-cost and high-quality products in a fast, efficient, and predictable way [15].

As discussed earlier, OO promises a high level of reuse via the inheritance feature and the use of class libraries. Nevertheless, OO's practical reuse is not as supportive of the described strategic reuse initiatives as one might like to see, and even the full and uncompromising implementation of OO does not guarantee the satisfaction of any aspects of the mentioned, reuse-centered corporate architecture initiatives.

Continuing Stream of Requirement Changes (Risk B6)

This risk is caused by customer behavior, and the use of OO is not a contributing factor. On the contrary, as it was pointed out in Risk B2, OO architectural considerations, encapsulation, and data hiding increase the developed system's resiliency to requirements volatility.

Shortfalls in Externally Performed Tasks (Risk B7)

Risk B7 is caused by contractor behavior, and the use of OO does not play any role. Nevertheless, similar to B6, the presence of M1 and M2 OO concepts is an excellent mitigating factor when these kinds of problems arise.

Straining Computer-Science Capabilities (Risk B8)

The appeal of the concepts M1-M4 (see Figure 2), which are theoretical in nature, inspires system architects to use OO in designing complex systems. Concepts M5-M7 are related to implementation, and their role is to enable and facilitate using the theoretical concepts. This risk item refers to the persistent tension between the theoretical concepts and their implementation, and the delicate balance that must be maintained among programming languages, developing environments, and analysis/design methods.

The viability and feasibility of all these elements have to be continually verified against the developed system's architecture. A recent example is the introduction of a promising new programming technique called Aspect-Oriented Programming (AOP). According to Gregor Kiczales, one of the principal developers of AOP, integrating AOP with OO development environments is difficult [16]. A standard development environment would have facilities for structure browsing, smart editing, refactoring, building, testing, and debugging, but it does not have a way to represent and directly manipulate AOP-specific constructs.

Summary

A systematic approach was presented to identify risks in OO development. The fundamental concepts of OO were introduced and matched against a well-known, methodology-neutral list of software risks. This dissection of OO concepts allows project managers to more completely understand the cost/benefit aspects of applying OO, and to align their project management strategies better with the organization's business goals.◆

Acknowledgements

This work would not have been possible without assistance from the following people and organizations:

- Reviewers: Richard J. Adams, Sergio Alvarado, Suellen Eslinger, and Joanne Tagami all with The Aerospace Corporation, and Scott A. Whitmire at ODS Software, Inc.
- Sponsor: Michael Zambrana, U.S. Air Force Space and Missile Center.
- Funding Source: Mission-Oriented Investigation and Experimentation Research Program, Software Acquisition Task.

A version of this article was presented at the 2004 Pacific Northwest Software Quality Conference (2004 PNSQC).

References

1. Glass, R.L. "The Naturalness of Object Orientation: Beating a Dead Horse?" IEEE Software May/June 2002.
2. Meyer, B. Object Success. Prentice Hall PTR, 1995.
3. The Institute of Electrical and Electronics Engineers. IEEE-STD-1540-2001 – Standard for Software Life Cycle Processes-Risk Management. New York: IEEE, 2001.
4. Merson, P. "Managing J2EE Risks." Software Development July 2004.
5. Moynihan, T. "How Experienced Project Managers Assess Risk." IEEE Software, May/June 1997.
6. Boehm, B. IEEE Tutorial on Software Risk Management. IEEE Computer Society Press, 1989.
7. Boehm, B. "Software Risk Management: Overview and Recent Developments." 17th International Forum on COCOMO and Software Cost Modeling. Los Angeles, CA, Oct. 2002.
8. Fichman, R., and C. Kemerer. "The Assimilation of Software Process Innovations: An Organizational Learning Perspective." Management Science, 1997.
9. Feiman, J. Migrating Developers to Java: Is It Worth the Cost and Risks? Stanford, CT: Gartner, 2000.
10. Flanagan, E.B. "Risky Business." C++ Report Mar.-Apr. 1995.
11. Binder, R.V. "Object-Oriented Testing: Myth and Reality." Object Magazine May 1995.
12. Smith, C.U. Performance Engineering of Software Systems. Addison-Wesley, 1990.
13. Sandén, B. "Coping with Java Threads." IEEE Computer Apr. 2004.
14. Jacobson, I., et al. Software Reuse. ACM Press, 1997.
15. Northrop, L.M. "A Practical Look at Software Product Lines." CASCON 2003, Ontario, CA, Oct. 2003.
16. Kiczales, G., and M. Kersten. "Show Me the Structure." Software Development Apr. 2000.

Notes

1. Please note that the M1-M11 numbering of concepts did not originate from Meyer; it was introduced by the author to facilitate the mapping process.
2. The Software Engineering Institute is

a federally funded research and development organization at Carnegie Mellon University, Pittsburgh, Pa.

3. Risk profile: A chronological record of a risk's current and historical state information [3].
4. Gold-plating is a popular software management term for implementing features by software engineers that go beyond the scope of actual requirements.
5. Consider the book's subtitle: "A manager's guide to object orientation, its impact on the corporation and its use for reengineering the software process."
6. Black Box testing targets externally observable behavior that is produced from a given input, without using any implementation information.
7. Quality in short is fitness for purpose, the degree to which a system accomplishes its designated functions within constraint. It includes all the -ities, e.g., availability, reliability, security, safety, etc.

About the Author

Peter Hantos, Ph.D., is currently a senior engineering specialist in the Software Acquisition and Process Office of the Software Engineering subdivision at The Aerospace Corporation. He has more than 30 years of experience as a professor, researcher, software engineer, and manager. Previously as principal scientist at the Xerox Corporate Engineering Center, Hantos developed corporate-wide engineering processes for software-intensive systems, and as department manager, he directed all aspects of quality for several laser printer product lines. He is author of numerous technical papers and U.S. and international conference presentations. Hantos is a member of the Association for Computing Machinery and senior member of the Institute of Electrical and Electronics Engineers. He has a Master of Science and doctorate degree in electrical engineering from the Budapest Institute of Technology, Hungary.

The Aerospace Corporation
P.O. Box 92957 – M1/112
El Segundo, CA 90009-2957
Phone: (310) 336-1802
Fax: (310) 563-1160
E-mail: peter.hantos@aero.org

Web Sites

Risk Management
www.acq.osd.mil/io/se/risk_management/index.htm
This is the Department of Defense (DoD) risk management Web site. The Systems Engineering group within the interoperability organization formed a working group of representatives from the services and other DoD agencies involved in systems acquisition to assist in the evaluation of the DoD's approach to risk management. The group will continue to provide a forum that provides program managers with the latest tools and advice on managing risk.

Software Technology Support Center
www.stsc.hill.af.mil
The Software Technology Support Center is an Air Force organization established to help other U.S. government organizations identify, evaluate, and adopt technologies to improve the quality of their software products, efficiency in producing them, and to accurately predict the cost and schedule of their delivery.

Software Program Managers Network
www.spmn.com
The Software Program Managers Network (SPMN) is sponsored by the deputy under secretary of defense for Science and Technology, Software Intensive Systems Directorate. It seeks out proven industry and government software best practices and conveys them to managers of large-scale DoD software-intensive acquisition programs. SPMN provides consulting, on-site program assessments, project risk assessments, software tools, and hands-on training.

Software Engineering Institute
www.sei.cmu.edu
The Software Engineering Institute (SEI) is a federally funded research and development center sponsored by the Department of Defense to provide leadership in advancing the state of the practice of software engineering to improve the quality of systems that depend on software. SEI helps organizations and individuals improve their software engineering management practices. The site features complete information on models the SEI is currently involved in developing, expanding, or maintaining, including the Capability Maturity Model® Integration, Capability Maturity Model® for Software, Software Acquisition Capability Maturity Model®, Systems Engineering Capability Maturity Model®, and more.

Project Management Institute
www.pmi.org
The Project Management Institute (PMI) claims to be the world's leading not-for-profit project management professional association. PMI provides global leadership in the development of standards for the practice of the project management profession throughout the world.

The Software Productivity Consortium
www.software.org
The Software Productivity Consortium is a nonprofit partnership of industry, government, and academia. It develops processes, methods, tools, and supporting services to help members and affiliates build high-quality, component-based systems, and continuously advance their systems and software engineering maturity pursuant to the guidelines of all of the major process and quality frameworks. Based on the members' collective needs, its technical program builds on current best practices and information technologies to create project-ready processes, methods, training, tools, and supporting services for systems and software development.

INCOSE
www.incose.org
The International Council on Systems Engineering (INCOSE) was formed to develop, nurture, and enhance the interdisciplinary approach to enable the realization of successful systems. INCOSE works with industry, academia, and government in these ways: provides a focal point for disseminating systems engineering knowledge, promotes collaboration in systems engineering education and research, assures the establishment of professional standards for integrity in the practice of systems engineering, and encourages governmental and industrial support for research and educational programs to improve the systems engineering process and its practices.